Môn Mẫu Thiết Kế Phần Mềm

Lớp Chiều Thứ 4

— * —

# BÁO CÁO CUỐI KỲ 10 MẪU THIẾT KẾ HÀNH VI

GVHD: Th.S Nguyễn Minh Đạo

SVTH: Nguyễn Trung Nhân, 21110266

Nội dung

# 1. Strategy pattern

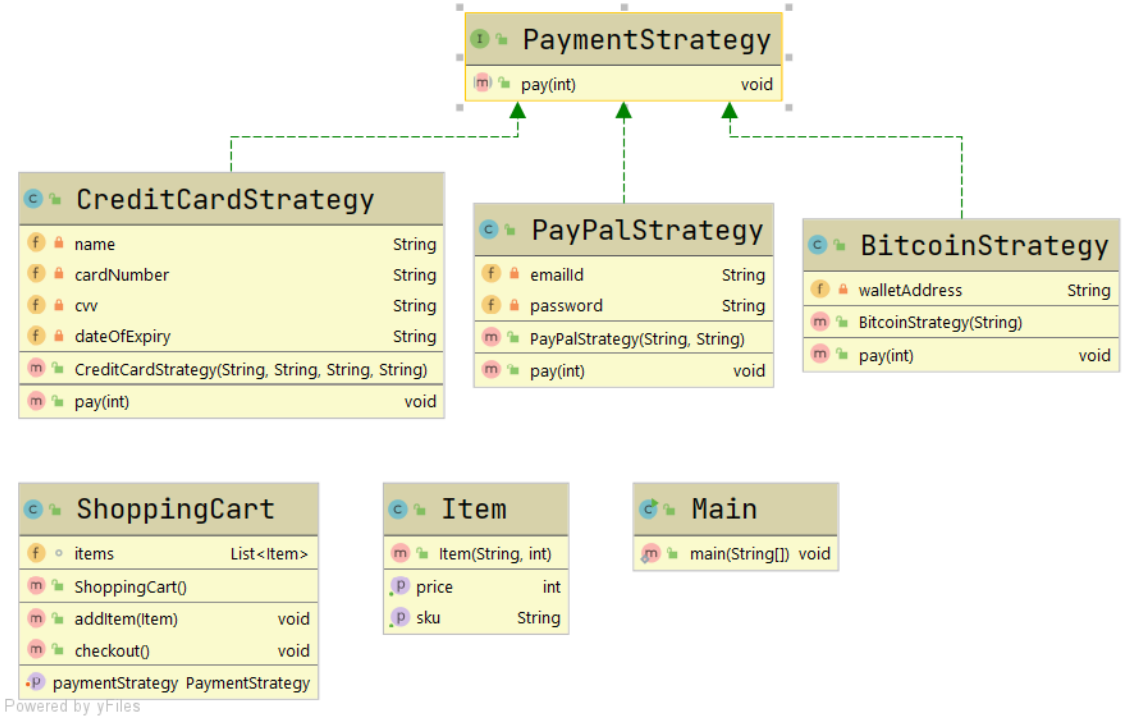
## 1.1. Theory

Ý nghĩa của Strategy Pattern là giúp tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng. Sau đó tạo ra một tập hợp các thuật toán để xử lý chức năng đó và lựa chọn thuật toán nào mà chúng ta thấy đúng đắn nhất khi thực thi chương trình. Mẫu thiết kế này thường được sử dụng để thay thế cho sự kế thừa, khi muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.

Các thành phần tham gia Strategy Pattern:

- Strategy : định nghĩa các hành vi có thể có của một Strategy.
- Concrete Strategy : cài đặt các hành vi cụ thể của Strategy.
- Context : chứa một tham chiếu đến đối tượng Strategy và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Strategy thực hiện.

## 1.2. Class diagram



## 1.3. Build and test

### 1.3.1. Bước 1: Chuẩn bị mẫu strategy

```
package vn.nhannt.fhqx.payment;

public interface PaymentStrategy {
   public void pay(int amount);
}
```

```
package vn.nhannt.fhqx.payment;

public class CreditCardStrategy implements PaymentStrategy {
   // fields
   private String name;
   private String cardNumber;
   private String cvv; // Card Verification Code
   private String dateOfExpiry;

   // constructor
     public CreditCardStrategy(String name, String cardNumber, String cvv,
String dateOfExpiry) {
       this.name = name;
       this.cardNumber = cardNumber;
       this.cvv = cvv;
       this.dateOfExpiry = dateOfExpiry;
   }

   @Override
   public void pay(int amount) {
       System.out.println("begin");
       System.out.println("\t status: 200");
       System.out.println("\t message: payment with credit/ debit card");
       System.out.println("\t data:");
       System.out.println("\t\t card holder: " + name);
       System.out.println("\t\t amount: " + amount);
       System.out.println("end");
   }

}
```

```java
package vn.nhannt.fhqx.payment;

public class PayPalStrategy implements PaymentStrategy {
    private String emailId;
    private String password;

    public PayPalStrategy(String emailId, String password) {
        this.emailId = emailId;
        this.password = password;
    }

    @Override
    public void pay(int amount) {
        System.out.println("begin");
        System.out.println("\t status: 200");
        System.out.println("\t message: payment with paypal");
        System.out.println("\t data:");
        System.out.println("\t\t email: " + emailId);
        System.out.println("\t\t amount: " + amount);
        System.out.println("end");
    }
}
```

```java
package vn.nhannt.fhqx.payment;

public class BitcoinStrategy implements PaymentStrategy{
    // fields
    String walletAddress; // a unique alphanumeric string

    // constructor
    public BitcoinStrategy(String walletAddress) {
        this.walletAddress = walletAddress;
    }

    // method
    @Override
    public void pay(int amount) {
        System.out.println("begin");
```

```
        System.out.println("\t status: 200");
        System.out.println("\t message: payment with bitcoin");
        System.out.println("\t data:");
        System.out.println("\t\t wallet address: " + walletAddress);
        System.out.println("\t\t amount: " + amount);
        System.out.println("end");
    }
}
```

### 1.3.2. Bước 2: Sử dụng mẫu strategy

```
package vn.nhannt.fhqx.shopping;

public class Item {
    // fields
    private String sku;
    private int price;

    // constructor
    public Item(String sku, int price) {
        this.sku = sku;
        this.price = price;
    }

    // properties
    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }
}
```

```
package vn.nhannt.fhqx.shopping;

import vn.nhannt.fhqx.payment.PaymentStrategy;
```

```java
import java.util.ArrayList;
import java.util.List;

public class ShoppingCart {
    // fields
    private List<Item> items; //List of items
    private PaymentStrategy paymentStrategy;

    // constructor
    public ShoppingCart(){
        this.items=new ArrayList<>();
    }

    // method
    public void addItem(Item item){
        this.items.add(item);
    }
    public int checkout(){
        int sum = 0;
        for(Item item : items){
            sum += item.getPrice();
        }
        return sum;
    }

    // properties
    public PaymentStrategy getPaymentStrategy() {
        return paymentStrategy;
    }

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }
}
```

### 1.3.3. Bước 3: Kiểm thử

```java
package vn.nhannt.fhqx.tester;
```

```java
import vn.nhannt.fhqx.payment.BitcoinStrategy;
import vn.nhannt.fhqx.payment.CreditCardStrategy;
import vn.nhannt.fhqx.payment.PayPalStrategy;
import vn.nhannt.fhqx.payment.PaymentStrategy;
import vn.nhannt.fhqx.shopping.Item;
import vn.nhannt.fhqx.shopping.ShoppingCart;

public class Main {
   public static void main(String[] args) {
       // create base items
       Item item1 = new Item("6841",40);
       Item item2 = new Item("6842",60);
       Item item3 = new Item("6843",80);

       // client 1
       ShoppingCart cart1 = new ShoppingCart();
       // client 1 add to cart
       cart1.addItem(item1);
       cart1.addItem(item2);
       // client 1 pay with credit card
         PaymentStrategy CARD = new CreditCardStrategy("Nguyen Trung Nhan",
"121314151617l8", "120", "02/29");
       cart1.setPaymentStrategy(CARD);
       cart1.getPaymentStrategy().pay(cart1.checkout());

       // client 2
       ShoppingCart cart2 = new ShoppingCart();
       // client 2 add to cart
       cart2.addItem(item2);
       cart2.addItem(item3);
       // client 2 pay with credit card
          PaymentStrategy PAYPAL = new PayPalStrategy("nhannt@outlook.com",
"123456");
       cart2.setPaymentStrategy(PAYPAL);
       cart2.getPaymentStrategy().pay(cart2.checkout());

       // client 3
       ShoppingCart cart3 = new ShoppingCart();
```

```
        // client 3 add to cart
        cart3.addItem(item3);
        cart3.addItem(item1);
        // client 3 pay with credit card
                                    PaymentStrategy     BITCOIN     =     new
BitcoinStrategy("12131415161718191O1a1b1c1d1e1f");
        cart3.setPaymentStrategy(BITCOIN);
        cart3.getPaymentStrategy().pay(cart3.checkout());



    }
}
```

#### 1.3.4. Kết quả:

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\idea
begin
     status: 200
     message: payment with credit/ debit card
     data:
         card holder: Nguyen Trung Nhan
         amount: 100
end
begin
     status: 200
     message: payment with paypal
     data:
         email: nhannt@outlook.com
         amount: 140
end
begin
     status: 200
     message: payment with bitcoin
     data:
         wallet address: 1213141516171819101a1b1c1d1e1f
         amount: 120
end

Process finished with exit code 0
```

# 2. Mediator pattern

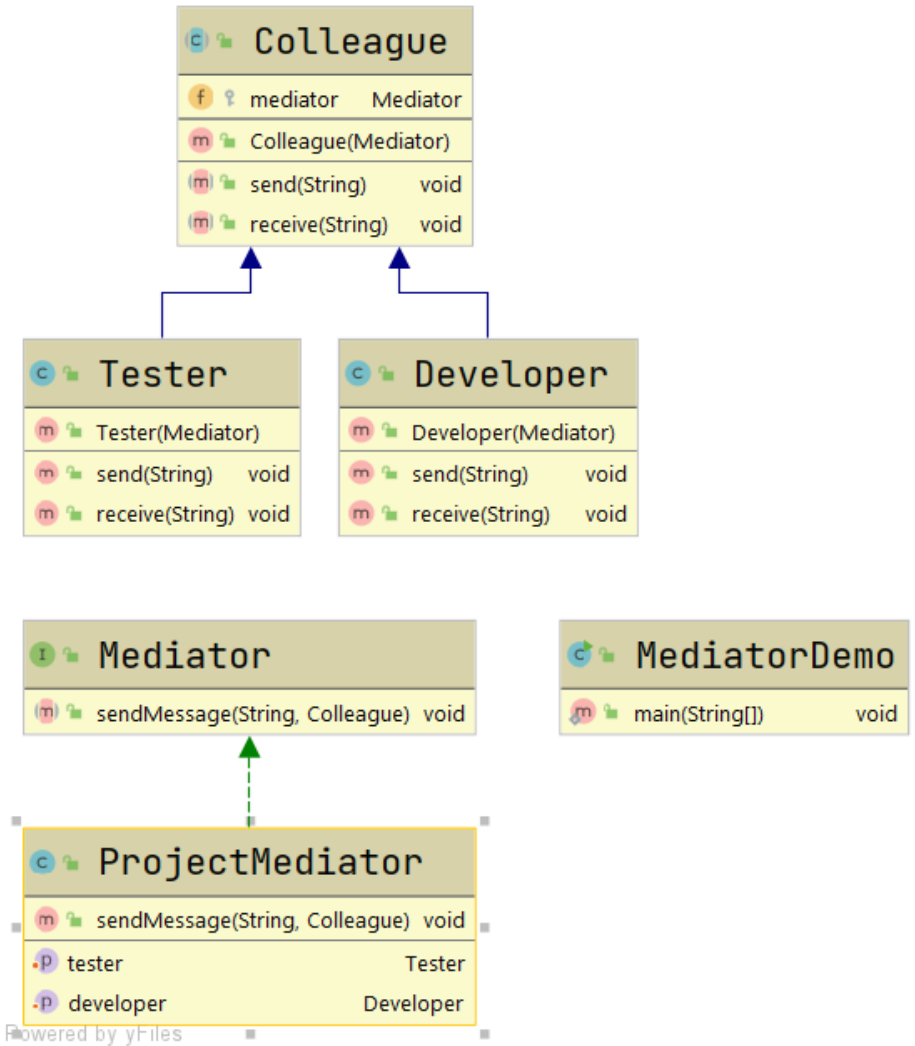
## 2.1. Theory

Mediator Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Mediator có nghĩa là người trung gian. Pattern này nói rằng "Định nghĩa một đối tượng gói gọn cách một tập hợp các đối tượng tương tác. Mediator thúc đẩy sự khớp nối lỏng lẻo (loose coupling) bằng cách ngăn không cho các đối tượng đề cập đến nhau một cách rõ ràng và nó cho phép bạn thay đổi sự tương tác của họ một cách độc lập".

Các thành phần tham gia Mediator Pattern:

- Colleague : là một abstract class, giữ tham chiếu đến Mediator object.
- Concrete Colleague : cài đặt các phương thức của Colleague. Giao tiếp thông qua Mediator khi cần giao tiếp với Colleague khác.
- Mediator : là một interface, định nghĩa các phương thức để giao tiếp với các Colleague object.
- Concrete Mediator : cài đặt các phương thức của Mediator, biết và quản lý các Colleague object.

## 2.2. Class diagram



## 2.3. Build and test

### 2.3.1. Bước 1: Chuẩn bị mẫu

```
package vn.nhannt.mediator;

public interface Mediator {
    void sendMesage(String message, Colleague sender);
}
```

```
package vn.nhannt.mediator;

public class ProjectMediator  implements Mediator{
    private Tester tester;
    private Developer developer;

    public Tester getTester() {
        return tester;
    }

    public void setTester(Tester tester) {
        this.tester = tester;
    }

    public Developer getDeveloper() {
        return developer;
    }

    public void setDeveloper(Developer developer) {
        this.developer = developer;
    }

    @Override
    public void sendMesage(String message, Colleague sender) {
        if(sender.equals(tester)) {
            developer.receive(message);
        } else if (sender.equals(developer)) {
```

```
            tester.receive(message);
        }
    }
}
```

### 2.3.2. Bước 2: Sử dụng mẫu

```
package vn.nhannt.mediator;

public abstract class Colleague {
    protected Mediator mediator;
    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }

    public abstract void send(String message);

    public abstract void receive(String message);
}
```

```
package vn.nhannt.mediator;

public class Developer extends Colleague{
    public Developer(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        System.out.println("Developer sends: " + message);
        mediator.sendMesage(message, this);
    }

    @Override
    public void receive(String message) {
        System.out.println("Developer received: " + message);
        // mediator.sendMesage(message, this);
    }
```

```
}
```

```
package vn.nhannt.mediator;

public class Tester extends Colleague{
    public Tester(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        System.out.println("Tester sends: " + message);
        mediator.sendMesage(message, this);

    }

    @Override
    public void receive(String message) {
        System.out.println("Tester received: " + message);
        // mediator.sendMesage(message, this);
    }
}
```

### 2.3.3. Bước 3: Kiểm thử

```
package vn.nhannt.mediator;

public class Main {
    public static void main(String[] args) {
        ProjectMediator projectMediator = new ProjectMediator();
        Tester tester = new Tester(projectMediator);
        Developer developer = new Developer(projectMediator);

        projectMediator.setTester(tester);
        projectMediator.setDeveloper(developer);

        tester.send("Có bug rồi kìa");
        developer.send("Fix rồi đó");
```

```
    }
}
```

#### 2.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\id
Tester sends: Có bug rồi kìa
Developer received: Có bug rồi kìa
Developer sends: Fix rồi đó
Tester received: Fix rồi đó

Process finished with exit code 0
```

# 3. State pattern

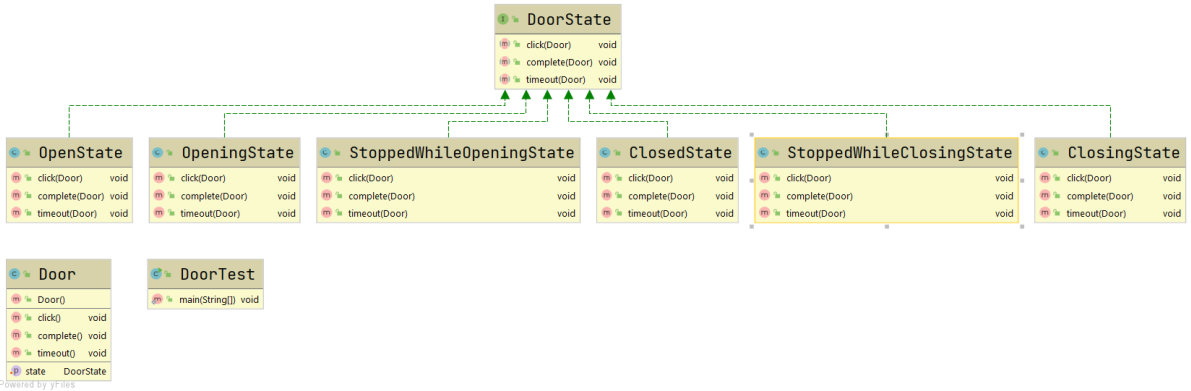
## 3.1. Theory

State Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

Các thành phần tham gia State Pattern:

- Context : được sử dụng bởi Client. Client không truy cập trực tiếp đến State của đối tượng. Lớp Context này chứa thông tin của Concrete State object, cho hành vi nào tương ứng với trạng thái nào hiện đang được thực hiện.
- State : là một interface hoặc abstract class xác định các đặc tính cơ bản của tất cả các đối tượng Concrete State. Chúng sẽ được sử dụng bởi đối tượng Context để truy cập chức năng có thể thay đổi.
- Concrete State : cài đặt các phương thức của State. Mỗi Concrete State có thể thực hiện logic và hành vi của riêng nó tùy thuộc vào Context.

## 3.2. Class diagram




## 3.3. Build and test

### 3.3.1. Bước 1: Chuẩn bị mẫu

```
package vn.nhannt.strate;

public interface DoorState {
    public void click(Door door);
    public void complete(Door door);

    public void timeout(Door door);
}
```

```
package vn.nhannt.strate;

public class OpenState implements DoorState{
    @Override
    public void click(Door door) {
        System.out.println("Click open state...");
    }

    @Override
    public void complete(Door door) {
        System.out.println("Complete open state...");
    }

    @Override
```

```
    public void timeout(Door door) {
        System.out.println("Time out open state...");
    }
}
```

```
package vn.nhannt.strate;

public class OpeningState implements DoorState{
    @Override
    public void click(Door door) {
        System.out.println("Click opening state...");
    }

    @Override
    public void complete(Door door) {
        System.out.println("Complete opening state...");
    }

    @Override
    public void timeout(Door door) {
        System.out.println("Time out opening state...");
    }
}
```

```
package vn.nhannt.strate;

public class ClosingState implements DoorState {
    @Override
    public void click(Door door) {
        System.out.println("Click closing state...");
    }

    @Override
    public void complete(Door door) {
        System.out.println("Complete closing state...");
    }

    @Override
    public void timeout(Door door) {
```

```
        System.out.println("Timeout closing state...");
    }
}
```

```
package vn.nhannt.strate;

public class ClosedState implements DoorState{
    @Override
    public void click(Door door) {
        System.out.println("Click closed state...");
    }

    @Override
    public void complete(Door door) {
        System.out.println("Complete closed state...");
    }

    @Override
    public void timeout(Door door) {
        System.out.println("Timeout closed state...");
    }
}
```

```
package vn.nhannt.strate;

public class StoppedWhileClosingState implements DoorState{
    @Override
    public void click(Door door) {
        System.out.println("Click stopped while closing state...");
    }

    @Override
    public void complete(Door door) {
        System.out.println("Complete stopped while closing state...");
    }

    @Override
    public void timeout(Door door) {
        System.out.println("Timeout stopped while closing state...");
```

```
    }
}
```

```
package vn.nhannt.strate;

public class StoppedWhileOpeningState implements DoorState{
    @Override
    public void click(Door door) {
        System.out.println("Click stopped while opening state...");
    }

    @Override
    public void complete(Door door) {
        System.out.println("Complete stopped while opening state...");
    }

    @Override
    public void timeout(Door door) {
        System.out.println("Time out stopped while opening state...");
    }
}
```

### 3.3.2. Bước 2: Sử dụng mẫu

```
package vn.nhannt.strate;

public interface DoorState {
    public void click(Door door);
    public void complete(Door door);

    public void timeout(Door door);
}
```

### 3.3.3. Bước 3: Kiểm thử

```
package vn.nhannt.strate;
```

```java
public class DoorTest {
    public static void main(String[] args) {
        //
        Door door = new Door();
        DoorState state1 = new OpenState();
        door.setState(state1);
        door.click();
        door.complete();
        door.timeout();
        //
        System.out.println();
        //
        DoorState state2 = new OpenState();
        door.setState(state2);
        door.click();
        door.complete();
        door.timeout();
    }
}
```

### 3.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\idea
Door clicking...
Door completing...
Door timeout

Door clicking...
Door completing...
Door timeout

Process finished with exit code 0
```

# 4. Chain of responsibility pattern

## 4.1. Theory

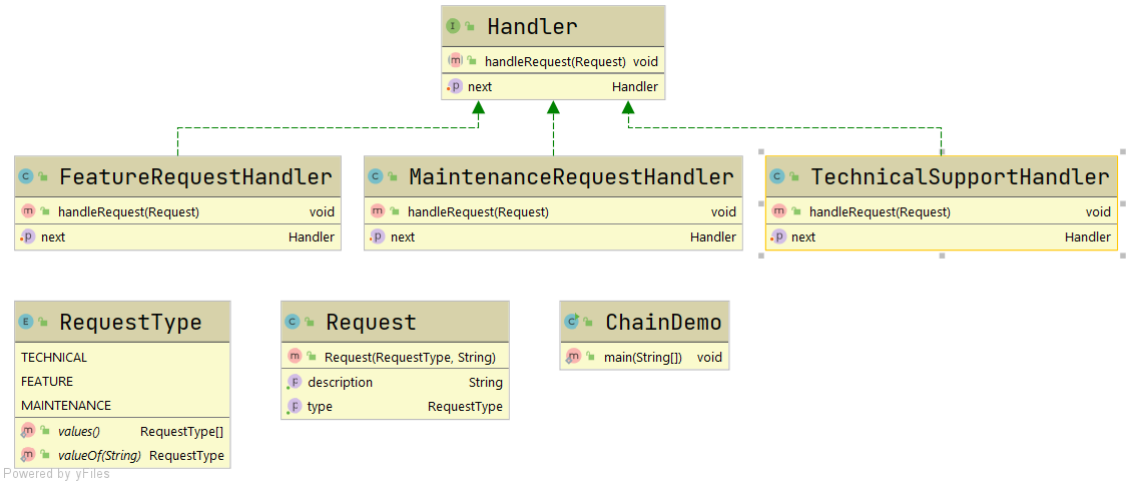
Chain of Responsibility (COR) là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern).

Chain of Responsibility cho phép một đối tượng gửi một yêu cầu nhưng không biết đối tượng nào sẽ nhận và xử lý nó. Điều này được thực hiện bằng cách kết nối các đối tượng nhận yêu cầu thành một chuỗi (chain) và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.

Các thành phần tham gia mẫu Chain of Responsibility:

- Handler : định nghĩa 1 interface để xử lý các yêu cầu. Gán giá trị cho đối tượng successor (không bắt buộc).
- Concrete Handler : xử lý yêu cầu. Có thể truy cập đối tượng successor (thuộc class Handler). Nếu đối tượng Concrete Handler không thể xử lý được yêu cầu, nó sẽ gửi lời yêu cầu cho successor của nó.
- Client : tạo ra các yêu cầu và yêu cầu đó sẽ được gửi đến các đối tượng tiếp nhận.

## 4.2. Class diagram




## 4.3. Build and test

### 4.3.1. Bước 1: Chuẩn bị mẫu

```
package vn.nhannt.chainofResponsibility;

public abstract class Handler {
    protected Handler next;

    public Handler getNext() {
        return next;
    }

    public void setNext(Handler next) {
        this.next = next;
    }

    public abstract void handleRequest(Request request);


}
```

```
package vn.nhannt.chainofResponsibility;

public class FeatureRequestHandler extends Handler{

    @Override
```

```
    public void handleRequest(Request request) {
        if (request.getTYPE() == RequestType.FEATURE) {
            System.out.println("Handle feature request successfully");
        } else {
            System.out.println("Handle feature request unsuccessfully");
            if(this.next != null) {
                this.next.handleRequest(request);
            }
        }


    }
}
```

```
package vn.nhannt.chainofResponsibility;

public class MaintainRequestHandler extends Handler{
    @Override
    public void handleRequest(Request request) {

        if (request.getTYPE() == RequestType.MAINTENANCE) {
            System.out.println("Handle maintenance request successfully");
        } else {
            System.out.println("Handle maintenance request unsuccessfully");
            if(this.next != null) {
                this.next.handleRequest(request);
            }
        }


    }
}
```

```
package vn.nhannt.chainofResponsibility;

public class TechnicalSupportHandler extends Handler {

    @Override
    public void handleRequest(Request request) {
```

```
        if (request.getTYPE() == RequestType.TECHNICAL) {
            System.out.println("Handle technical request successfully");
        } else {
            System.out.println("Handle technical request unsuccessfully");
            if(this.next != null) {
                this.next.handleRequest(request);
            }
        }


    }
}
```

### 4.3.2. Bước 2: Sử dụng mẫu

```
package vn.nhannt.chainofResponsibility;

import java.util.ArrayList;
import java.util.List;

public enum RequestType {
    TECHNICAL, FEATURE, MAINTENANCE
    ;
}
```

```
package vn.nhannt.chainofResponsibility;

public class Request {
    private String description;
    private RequestType TYPE;

    public Request(String description, RequestType TYPE) {
        this.description = description;
        this.TYPE = TYPE;
    }

    public String getDescription() {
        return description;
```

```
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public RequestType getTYPE() {
        return TYPE;
    }

    public void setTYPE(RequestType TYPE) {
        this.TYPE = TYPE;
    }
}
```

### 4.3.3. Bước 3: Kiểm thử

```
package vn.nhannt.chainofResponsibility;

public class Main {
    public static void main(String[] args) {
        FeatureRequestHandler handlerFeature = new FeatureRequestHandler();
        MaintainRequestHandler handlerMaintain = new MaintainRequestHandler();
        TechnicalSupportHandler handlerTech = new TechnicalSupportHandler();

        handlerFeature.setNext(handlerMaintain);
        handlerMaintain.setNext(handlerTech);
        handlerTech.setNext(handlerFeature);

                Request request1 = new Request("Yeu cau tinh nang moi",
RequestType.FEATURE);
        Handler handler1 = handlerFeature;
        handler1.handleRequest(request1);

        System.out.println("\n=======================\n");
```

```
            Request request2 = new Request("Yeu cau tinh nang moi",
RequestType.FEATURE);
        Handler handler2 = handlerMaintain;
        handler2.handleRequest(request2);
    }
}
```

#### 4.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\idea
Handle feature request successfully

========================

Handle maintenance request unsuccessfully
Handle technical request unsuccessfully
Handle feature request successfully

Process finished with exit code 0
```

# 5. Template method pattern

## 5.1. Theory

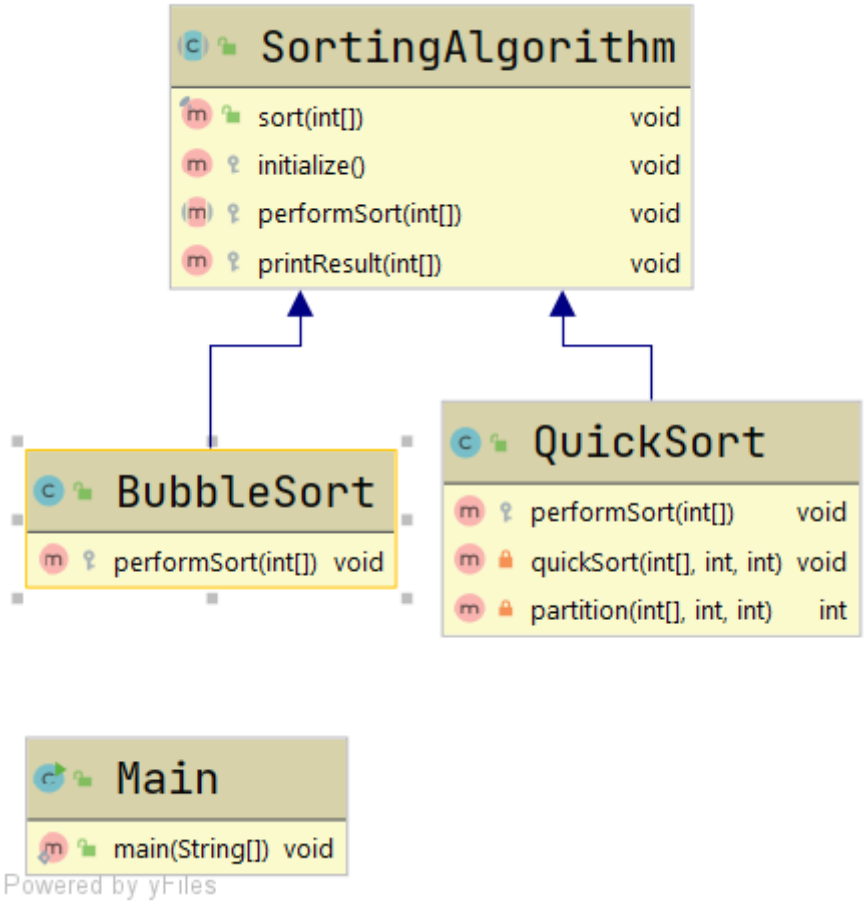
Template Method Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Pattern này nói rằng "Định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Mẫu Template Method cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán, mà không phải thay đổi cấu trúc thuật toán".

Các thành phần tham gia Template Method Pattern:

- Abstract Class: Định nghĩa các phương thức trừu tượng cho từng bước có thể được điều chỉnh bởi các lớp con. Cài đặt một phương thức duy nhất điều khiển thuật toán và gọi các bước riêng lẻ đã được cài đặt ở các lớp con.

- Concrete Class : là một thuật toán cụ thể, cài đặt các phương thức của Abstract Class. Các thuật toán này ghi đè lên các phương thức trừu tượng để cung cấp các triển khai thực sự. Nó không thể ghi đè phương thức duy nhất đã được cài đặt ở Abstract Class (Template Method).

## 5.2. Class diagram




## 5.3. Build and test

### 5.3.1. Bước 1: Chuẩn bị mẫu

```
package fhqx.template;

import java.util.Arrays;
import java.util.List;
```

```java
public abstract class SortingAlgorithm {
    // final method
    public final void sort(List<Integer> items) {
        initialize();
        performSort(items);
        printResult(items);
    }

    //
    protected abstract void performSort(List<Integer> items);

    protected void initialize() {
        System.out.println("Initializing ...");
    }

    protected void printResult(List<Integer> items) {
        System.out.println("Result: " + items + "\n");
    }
}
```

### 5.3.2. Bước 2: Sử dụng mẫu

```java
package fhqx.sort;

import fhqx.template.SortingAlgorithm;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class BubbleSort extends SortingAlgorithm {
    @Override
    protected void performSort(List<Integer> items) {
        System.out.println("Performing bubble sort ...");
        items.clear();
        items.add(1);
        items.add(2);
```

```
        items.add(3);
        items.add(4);
        items.add(5);

    }
}
```

```
package fhqx.sort;

import fhqx.template.SortingAlgorithm;

import java.util.List;

public class QuickSort extends SortingAlgorithm {
    @Override
    protected void performSort(List<Integer> items) {
        int partition = this.partition(items, 1, 5);
        this.quickSort(items, 1, 5);
    }

    private void quickSort(List<Integer> items, int num1, int num2) {
        System.out.println("Performing quick sort ...");
        items.clear();
        items.add(1);
        items.add(2);
        items.add(3);
        items.add(4);
        items.add(5);
    }

    private int partition(List<Integer> items, int num1, int num2) {
        return 3;
    }

}
```

### 5.3.3. Bước 3: Kiểm thử

```
package fhqx.tester;

import fhqx.sort.BubbleSort;
import fhqx.sort.QuickSort;
import fhqx.template.SortingAlgorithm;
import mypattern.body.AccountPage;
import mypattern.body.HomePage;
import mypattern.body.ItemPage;
import mypattern.template.PageTemplate;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> items = new ArrayList<>(Arrays.asList(5, 4, 3, 2, 1));
        // sort 1 - bubble
        SortingAlgorithm sort1 = new BubbleSort();
        sort1.sort(items);

        // page 2 - quick
        SortingAlgorithm sort2 = new QuickSort();
        sort2.sort(items);
    }
}
```

#### 5.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\ide
Initializing ...
Performing bubble sort ...
Result: [1, 2, 3, 4, 5]

Initializing ...
Performing quick sort ...
Result: [1, 2, 3, 4, 5]


Process finished with exit code 0
```

# 6. Visitor pattern

## 6.1. Theory

Visitor Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Visitor cho phép định nghĩa các thao tác (operations) trên một tập hợp các đối tượng (objects) không đồng nhất (về kiểu) mà không làm thay đổi định nghĩa về lớp (classes) của các đối tượng đó. Để đạt được điều này, trong mẫu thiết kế visitor ta định nghĩa các thao tác trên các lớp tách biệt gọi các lớp visitors, các lớp này cho phép tách rời các thao tác với các đối tượng mà nó tác động đến. Với mỗi thao tác được thêm vào, một lớp visitor tương ứng được tạo ra.

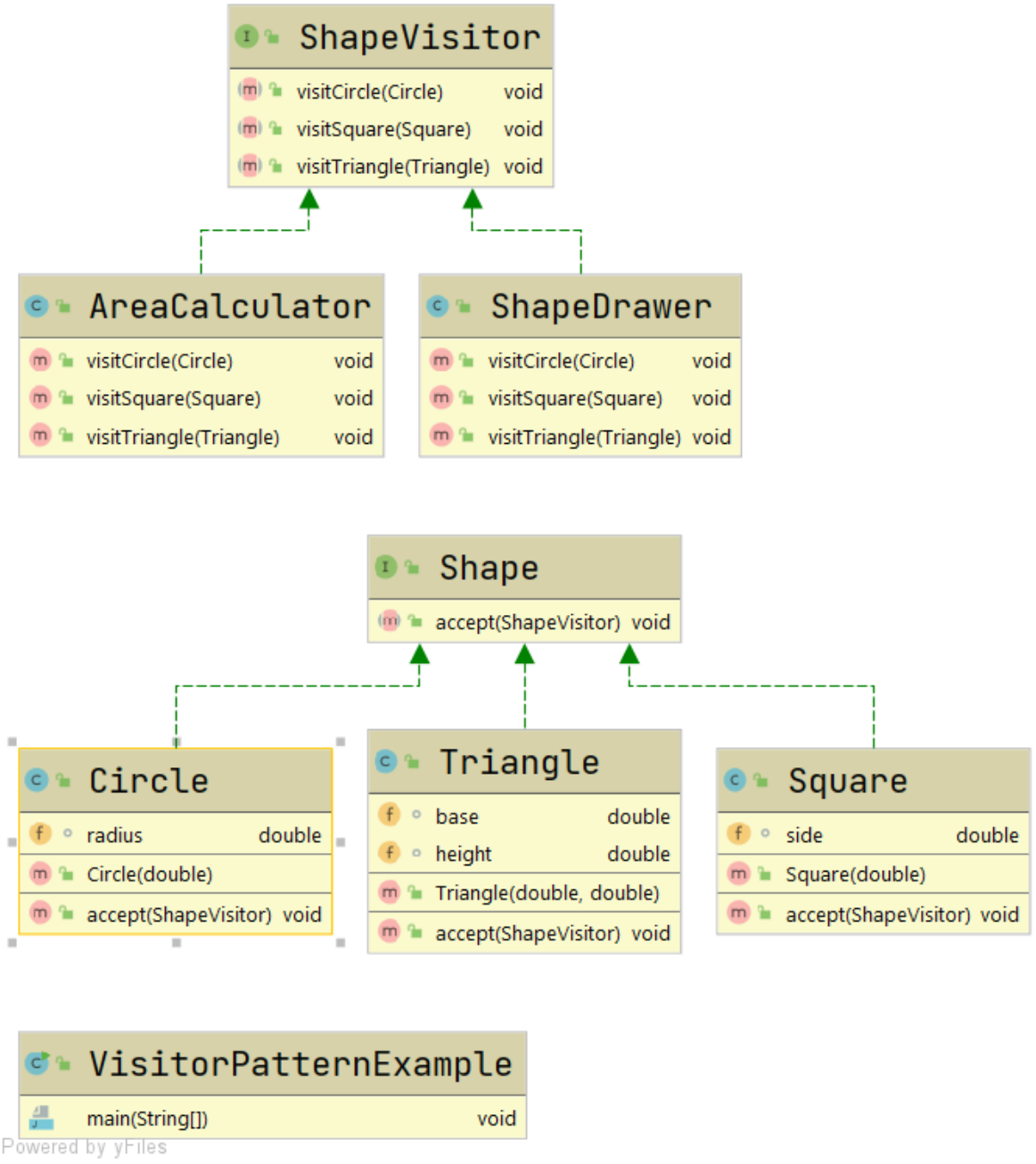
Các thành phần tham gia Visitor Pattern:

- Visitor :
    - Là một interface hoặc một abstract class được sử dụng để khai báo các hành vi cho tất cả các loại visitor.
    - Class này định nghĩa một loạt các các phương thức truy cập chấp nhận các Concrete Element cụ thể khác nhau làm tham số. Điều này sẽ hơi giống với cơ chế nạp chồng (overloading) nhưng các loại tham số nên

khác nhau do đó các hành vi hoàn toàn khác nhau. Các hành vi truy cập sẽ được thực hiện trên từng phần tử cụ thể trong cấu trúc đối tượng thông qua phương thức visit(). Loại phần tử cụ thể đầu vào sẽ quyết định phương thức được gọi.

- Concrete Visitor : cài đặt tất cả các phương thức abstract đã khai báo trong Visitor. Mỗi visitor sẽ chịu trách nhiệm cho các hành vi khác nhau của đối tượng.
- Element (Visible): là một thành phần trừu tượng, nó khai báo phương thức accept() và chấp nhận đối số là Visitor.
- Concrete Element (Concrete Visible): cài đặt phương thức đã được khai báo trong Element dựa vào đối số visitor được cung cấp.
- Object Structure : là một lớp chứa tất cả các đối tượng Element, cung cấp một cơ chế để duyệt qua tất cả các phần tử. Cấu trúc đối tượng này có thể là một tập hợp (collection) hoặc một cấu trúc phức tạp giống như một đối tượng tổng hợp (composite).
- Client : không biết về Concrete Element và chỉ gọi phương thức accept() của Element.

## 6.2. Class diagram




## 6.3. Build and test

### 6.3.1. Bước 1: Chuẩn bị mẫu

```
package vn.nhannt.visitorpattern;

public interface ShapeVisitor {
```

```
    public void visitCircle(Circle shape);
    public void visitTriangle(Triangle shape);
    public void visitSquare(Square shape);
}
```

```
package vn.nhannt.visitorpattern;

public class AreaCalculator implements ShapeVisitor{
    @Override
    public void visitCircle(Circle shape) {
        System.out.println("Circle Area: " + shape.getRadius() * 3.14);
    }

    @Override
    public void visitTriangle(Triangle shape) {
                System.out.println("Triangle  Area:  "  +  shape.getBase()  *
shape.getHeight() * 1/2);
    }

    @Override
    public void visitSquare(Square shape) {
                 System.out.println("Square  Area:  "  +  shape.getSide()  *
shape.getSide());
    }
}
```

```
package vn.nhannt.visitorpattern;

public class ShapeDrawer implements ShapeVisitor{
    @Override
    public void visitCircle(Circle shape) {
        System.out.println("Draw Circle");
    }

    @Override
    public void visitTriangle(Triangle shape) {
        System.out.println("Draw Triangle");
    }
```

```
    @Override
    public void visitSquare(Square shape) {
        System.out.println("Draw Square");
    }
}
```

### 6.3.2. Bước 2: Sử dụng mẫu

```
package vn.nhannt.visitorpattern;

public interface Shape {
    public void accept(ShapeVisitor visitor);
}
```

```
package vn.nhannt.visitorpattern;

public class Circle implements Shape{
    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }


    @Override
    public void accept(ShapeVisitor visitor) {
        visitor.visitCircle(this);
    }
}
```

```java
package vn.nhannt.visitorpattern;

public class Square implements Shape{
    private double side;

    public double getSide() {
        return side;
    }

    public void setSide(double side) {
        this.side = side;
    }

    public Square(double side) {
        this.side = side;
    }

    @Override
    public void accept(ShapeVisitor visitor) {
        visitor.visitSquare(this);
    }
}
```

```java
package vn.nhannt.visitorpattern;

public class Triangle implements Shape {
    private double base;

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    private double height;

    public double getBase() {
```

```
        return base;
    }

    public void setBase(double base) {
        this.base = base;
    }

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    @Override
    public void accept(ShapeVisitor visitor) {
        visitor.visitTriangle(this);
    }
}
```

### 6.3.3. Bước 3: Kiểm thử

```
package vn.nhannt.visitorpattern;

public class VisitorPatternExample {
    public static void main(String[] args) {
        Shape shape1 = new Circle(5);
        ShapeVisitor visitor1 = new AreaCalculator();
        shape1.accept(visitor1);

        System.out.println();

        Shape shape2 = new Square(5);
        ShapeVisitor visitor2 = new ShapeDrawer();
        shape2.accept(visitor2);
    }
}
```

#### 6.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\idea
Circle Area: 15.700000000000001

Draw Square

Process finished with exit code 0
```

# 7. Iterator pattern

## 7.1. Theory

Iterator Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó được sử dụng để "Cung cấp một cách thức truy cập tuần tự tới các phần tử của một đối tượng tổng hợp, mà không cần phải tạo dựng riêng các phương pháp truy cập cho đối tượng tổng hợp này".

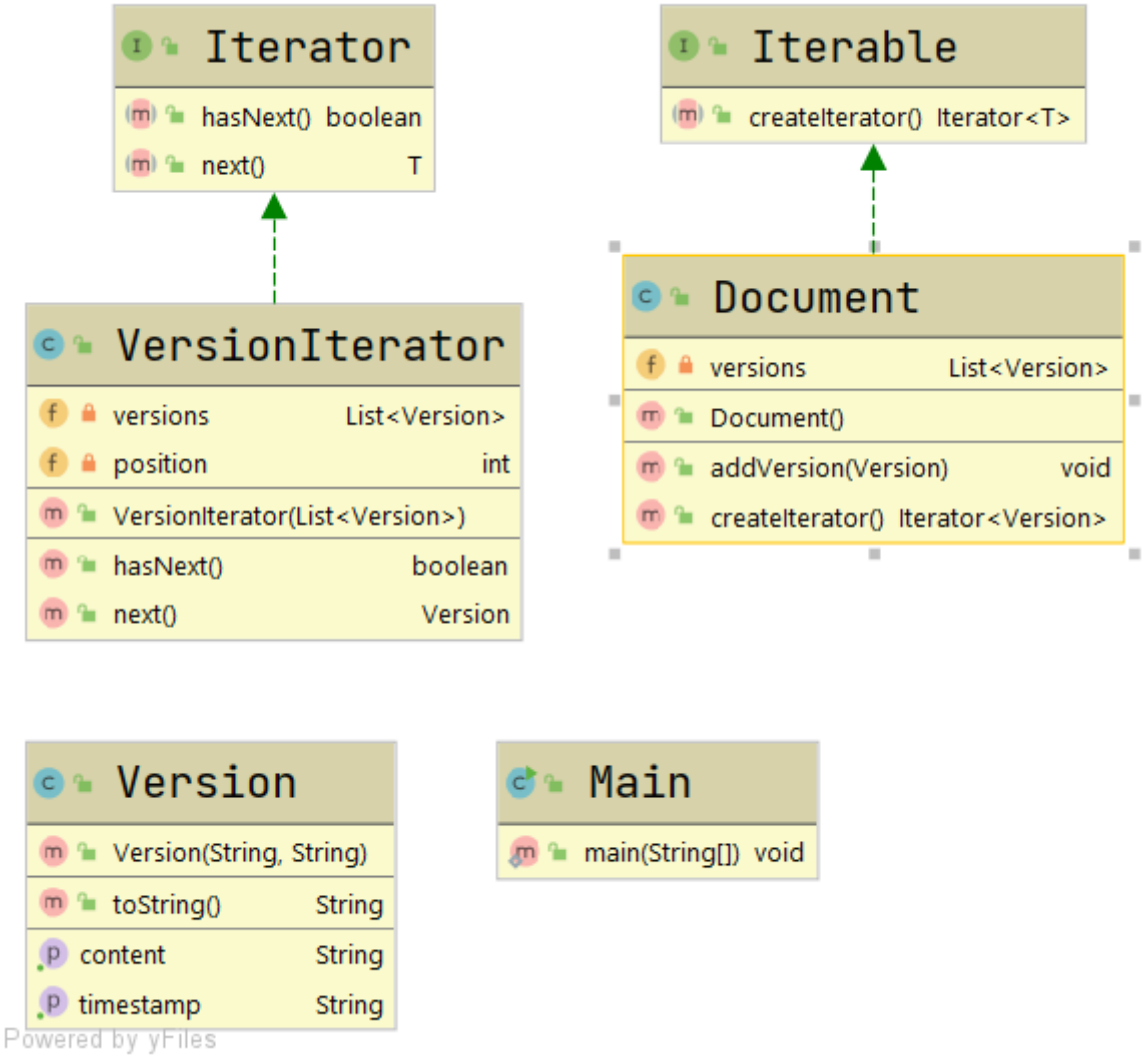
Nói cách khác, một Iterator được thiết kế cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn, mà không cần phải hiểu rõ về những chi tiết bên trong của những tập hợp này.

Các thành phần tham gia mẫu Iterator:

- Aggregate : là một interface định nghĩa định nghĩa các phương thức để tạo Iterator object.
- ConcreteAggregate : cài đặt các phương thức của Aggregate, nó cài đặt interface tạo Iterator để trả về một thể hiện của Concrete Iterator thích hợp.
- Iterator : là một interface hay abstract class, định nghĩa các phương thức để truy cập và duyệt qua các phần tử.
- Concrete Iterator : cài đặt các phương thức của Iterator, giữ index khi duyệt qua các phần tử.

- Client : đối tượng sử dụng Iterator Pattern, nó yêu cầu một iterator từ một đối tượng collection để duyệt qua các phần tử mà nó giữ. Các phương thức của iterator được sử dụng để truy xuất các phần tử từ collection theo một trình tự thích hợp.

## 7.2. Class diagram



## 7.3. Build and test

### 7.3.1. Bước 1: Chuẩn bị mẫu

```java
package vn.nhannt.iterator;

public interface Iterator<T> {
    public boolean hasNext();

    public T next();
}
```

```java
package vn.nhannt.iterator;

import java.util.List;

public class VersionIterator implements Iterator<Version> {
    List<Version> versions;
    int position;

    public VersionIterator(List<Version> versions) {
        this.versions = versions;
    }

    @Override
    public boolean hasNext() {
        while (position < versions.size()) {
            Version c = versions.get(position);
            if (c.getTimestamp().equals("8/5/24")) {
                return true;
            } else
                position++;
        }
        return false;
    }

    @Override
    public Version next() {
        Version version = versions.get(position);
        position++;
```

```
        return version;
    }
}
```

### 7.3.2. Bước 2: Sử dụng mẫu

```
package vn.nhannt.iterator;

public interface Iterable<T> {
    public Iterator<T> createIterator();
}
```

```
package vn.nhannt.iterator;

import java.util.ArrayList;
import java.util.List;

public class Document implements Iterable<Version>{
    private List<Version> versions;

    public Document() {
        versions = new ArrayList<>();
    }

    public void addIterator(Version version) {
        this.versions.add(version);
    }

    @Override
    public Iterator<Version> createIterator() {
        return new VersionIterator(versions);
    }
}
```

```
package vn.nhannt.iterator;

public class Version {
    private String content;
```

```
    private String timestamp;

    public Version(String content, String timestamp) {
        this.content = content;
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return "Version{" +
                "content='" + content + '\'' +
                ", timestamp='" + timestamp + '\'' +
                '}';
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }
}
```

### 7.3.3. Bước 3: Kiểm thử

```
package vn.nhannt.iterator;

public class Main {
    public static void main(String[] args) {
```

```
        Document versions = new Document();
        versions.addIterator(new Version("8.1.24", "8/1/24"));
        versions.addIterator(new Version("8.2.24", "8/2/24"));
        versions.addIterator(new Version("8.3.24", "8/3/24"));
        versions.addIterator(new Version("8.4.24", "8/4/24"));
        versions.addIterator(new Version("8.5.24", "8/5/24"));

        Iterator<Version> baseIterator = versions.createIterator();
        while (baseIterator.hasNext()) {
            Version c = baseIterator.next();
            System.out.println(c.toString());
        }
    }
}
```

#### 7.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\idea
Version{content='8.5.24', timestamp='8/5/24'}

Process finished with exit code 0
```

# 8. Command pattern

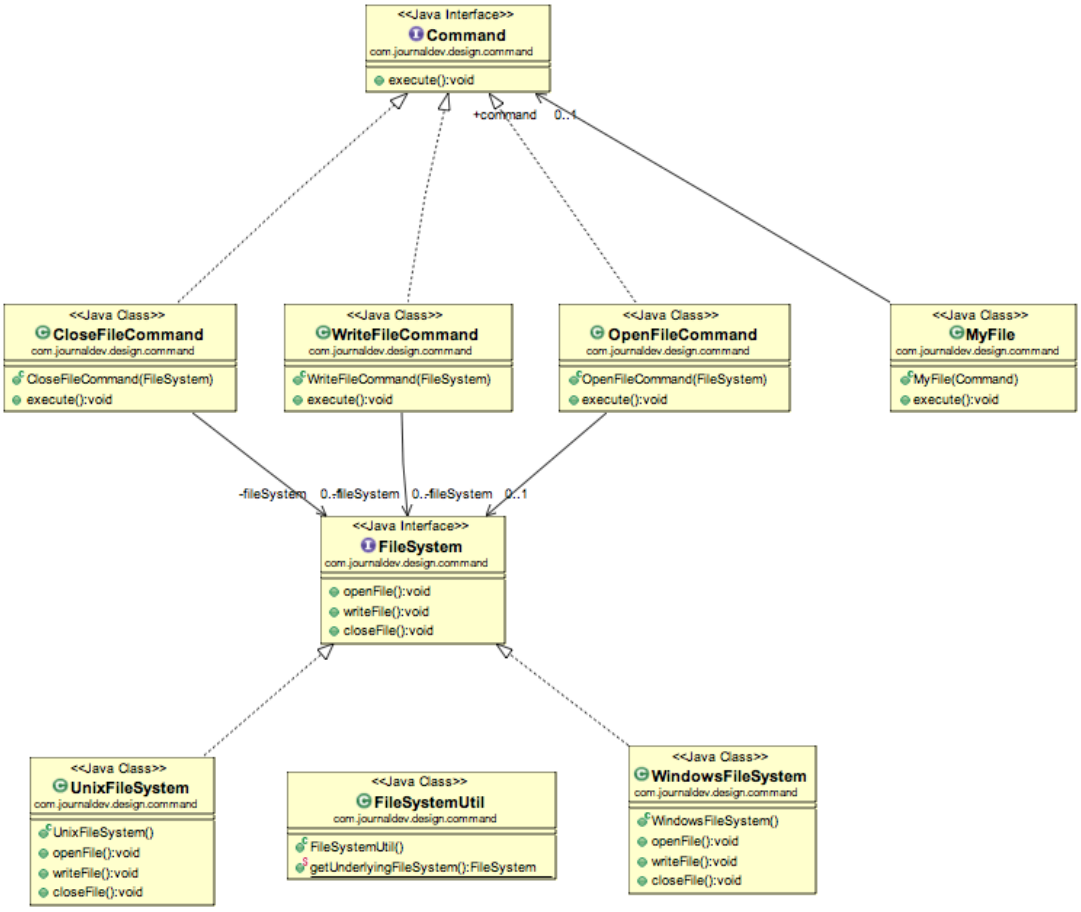
## 8.1. Theory

Trong mẫu lệnh, yêu cầu được gửi đến kẻ gọi và kẻ gọi sẽ chuyển nó đến đối tượng lệnh được đóng gói. Đối tượng lệnh chuyển yêu cầu đến phương thức thích hợp của Người nhận để thực hiện hành động cụ thể. Chương trình máy khách tạo đối tượng người nhận và sau đó đính kèm nó vào Lệnh. Sau đó, nó tạo đối tượng Invoker và đính kèm đối tượng Lệnh để thực hiện một hành động. Bây giờ khi chương trình máy khách thực hiện hành động, nó sẽ được xử lý dựa trên đối tượng lệnh và người nhận.

Các thành phần tham gia trong Command Pattern:

- Command: là một interface hoặc abstract class, chứa một phương thức trừu tượng thực thi (execute) một hành động (operation). Request sẽ được đóng gói dưới dạng Command.
- Concrete Command: là các implementation của Command. Định nghĩa một sự gắn kết giữa một đối tượng Receiver và một hành động. Thực thi execute() bằng việc gọi operation đang hoãn trên Receiver. Mỗi một Concrete Command sẽ phục vụ cho một case request riêng.
- Client: tiếp nhận request từ phía người dùng, đóng gói request thành Concrete Command thích hợp và thiết lập receiver của nó.
- Invoker: tiếp nhận Concrete Command từ Client và gọi execute() của Concrete Command để thực thi request.
- Receiver : đây là thành phần thực sự xử lý business logic cho case request. Trong phương execute() của Concrete Command chúng ta sẽ gọi method thích hợp trong Receiver

## 8.2. Class diagram




## 8.3. Build and test

### 8.3.1. Bước 1: Chuẩn bị mẫu

```
package vn.nhannt.command;

public interface Command {
    void execute();
}
```

```
package vn.nhannt.command;

public class CloseFileCommand implements Command {
    private FileSystemReceiver fileSystem;
```

```
    public CloseFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
    }
    @Override
    public void execute() {
        this.fileSystem.closeFile();
    }
}
```

```
package vn.nhannt.command;

public class OpenFileCommand implements Command {
    private FileSystemReceiver fileSystem;

    public OpenFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
    }
    @Override
    public void execute() {
        //open command is forwarding request to openFile method
        this.fileSystem.openFile();
    }
}
```

```
package vn.nhannt.command;

public class WriteFileCommand implements Command {
    private FileSystemReceiver fileSystem;

    public WriteFileCommand(FileSystemReceiver fs){
        this.fileSystem=fs;
    }
    @Override
    public void execute() {
        this.fileSystem.writeFile();
    }
}
```

### 8.3.2. Bước 2: Sử dụng mẫu

```
package vn.nhannt.command;

public interface FileSystemReceiver {
   void openFile();
   void writeFile();
   void closeFile();
}
```

```
package vn.nhannt.command;

public class UnixFileSystemReceiver implements FileSystemReceiver{
   @Override
   public void openFile() {
       System.out.println("Opening file in unix OS");
   }

   @Override
   public void writeFile() {
       System.out.println("Writing file in unix OS");
   }

   @Override
   public void closeFile() {
       System.out.println("Closing file in unix OS");
   }
}
```

```
package vn.nhannt.command;

public class WindowsFileSystemReceiver implements FileSystemReceiver {
   @Override
   public void openFile() {
       System.out.println("Opening file in Windows OS");

   }

   @Override
```

```
    public void writeFile() {
        System.out.println("Writing file in Windows OS");
    }

    @Override
    public void closeFile() {
        System.out.println("Closing file in Windows OS");
    }
}
```

```
package vn.nhannt.command;

public class FileSystemReceiverUtil {
    public static FileSystemReceiver getUnderlyingFileSystem(){
        String osName = System.getProperty("os.name");
        System.out.println("Underlying OS is:"+osName);
        if(osName.contains("Windows")){
            return new WindowsFileSystemReceiver();
        }else{
            return new UnixFileSystemReceiver();
        }
    }
}
```

```
package vn.nhannt.command;

public class FileInvoker {
    public Command command;

    public FileInvoker(Command c){
        this.command=c;
    }

    public void execute(){
        this.command.execute();
    }
}
```

### 8.3.3. Bước 3: Kiểm thử

```
package vn.nhannt.command;

public class Main {
    public static void main(String[] args) {
        //Creating the receiver object
                                                FileSystemReceiver      fs      =
FileSystemReceiverUtil.getUnderlyingFileSystem();

        //creating command and associating with receiver
        OpenFileCommand openFileCommand = new OpenFileCommand(fs);

        //Creating invoker and associating with Command
        FileInvoker file = new FileInvoker(openFileCommand);

        //perform action on invoker object
        file.execute();

        WriteFileCommand writeFileCommand = new WriteFileCommand(fs);
        file = new FileInvoker(writeFileCommand);
        file.execute();

        CloseFileCommand closeFileCommand = new CloseFileCommand(fs);
        file = new FileInvoker(closeFileCommand);
        file.execute();
    }
}
```

### 8.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\idea_
Underlying OS is:Windows 11
Opening file in Windows OS
Writing file in Windows OS
Closing file in Windows OS

Process finished with exit code 0
```

# 9. Memento pattern

## 9.1. Theory

Memento là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Memento là mẫu thiết kế có thể lưu lại trạng thái của một đối tượng để khôi phục lại sau này mà không vi phạm nguyên tắc đóng gói.

Dữ liệu trạng thái đã lưu trong đối tượng memento không thể truy cập bên ngoài đối tượng được lưu và khôi phục. Điều này bảo vệ tính toàn vẹn của dữ liệu trạng thái đã lưu.

Hoàn tác (Undo) hoặc ctrl + z là một trong những thao tác được sử dụng nhiều nhất trong trình soạn thảo văn bản (editor). Mẫu thiết kế Memento được sử dụng để thực hiện thao tác Undo. Điều này được thực hiện bằng cách lưu trạng thái hiện tại của đối tượng mỗi khi nó thay đổi trạng thái, từ đó chúng ta có thể khôi phục nó trong mọi trường hợp có lỗi.

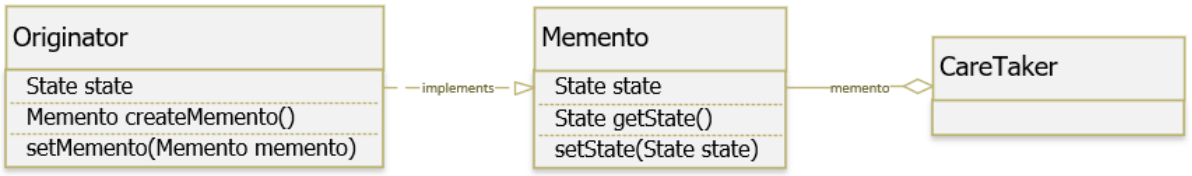
Các thành phần tham gia mẫu Memento:

- Originator : đại diện cho đối tượng mà chúng ta muốn lưu. Nó sử dụng memento để lưu và khôi phục trạng thái bên trong của nó.
- Caretaker : Nó không bao giờ thực hiện các thao tác trên nội dung của memento và thậm chí nó không kiểm tra nội dung. Nó giữ đối tượng memento và chịu trách nhiệm bảo vệ an toàn cho các đối tượng. Để khôi phục trạng thái trước đó, nó trả về đối tượng memento cho Originator.
- Memento : đại diện cho một đối tượng để lưu trữ trạng thái của Originator. Nó bảo vệ chống lại sự truy cập của các đối tượng khác ngoài Originator.
    - Lớp Memento cung cấp 2 interfaces: 1 interface cho Caretaker và 1 cho Originator. Interface Caretaker không được cho phép bất kỳ hoạt động hoặc bất kỳ quyền truy cập vào trạng thái nội bộ được lưu trữ bởi memento và do đó đảm bảo nguyên tắc đóng gói. Interface Originator

cho phép nó truy cập bất kỳ biến trạng thái nào cần thiết để có thể khôi phục trạng thái trước đó.

- Lớp Memento thường là một lớp bên trong của Originator. Vì vậy, originator có quyền truy cập vào các trường của memento, nhưng các lớp bên ngoài không có quyền truy cập vào các trường này

## 9.2. Class diagram




## 9.3. Build and test

### 9.3.1. Bước 1: Chuẩn bị mẫu

```
public static class Memento {
    private final String state;

    public Memento(String stateToSave) {
        state = stateToSave;
    }

    public String getSavedState() {
        return state;
    }
}
```

### 9.3.2. Bước 2: Sử dụng mẫu

```
public class Originator {
    private String state;
```

```
    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(this.state);
    }

    public void restoreFromMemento(Memento memento) {
        this.state = memento.getSavedState();
         System.out.println("Originator: State after restoring from Memento: "
+ state);
    }

    public static class Memento {
        private final String state;

        public Memento(String stateToSave) {
            state = stateToSave;
        }

        public String getSavedState() {
            return state;
        }
    }
}
```

### 9.3.3. Bước 3: Kiểm thử

```
package vn.nhannt.memento;

import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
```

```
        List<Originator.Memento> savedStates = new ArrayList<>(); // caretaker

        Originator originator = new Originator();
        originator.set("State #1");
        originator.set("State #2");
        savedStates.add(originator.saveToMemento());
        originator.set("State #3");
        savedStates.add(originator.saveToMemento());
        originator.set("State #4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

#### 9.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\idea
Originator: Setting state to State #1
Originator: Setting state to State #2
Originator: Saving to Memento.
Originator: Setting state to State #3
Originator: Saving to Memento.
Originator: Setting state to State #4
Originator: State after restoring from Memento: State #3

Process finished with exit code 0
```

# 10. Observer pattern

## 10.1. Theory

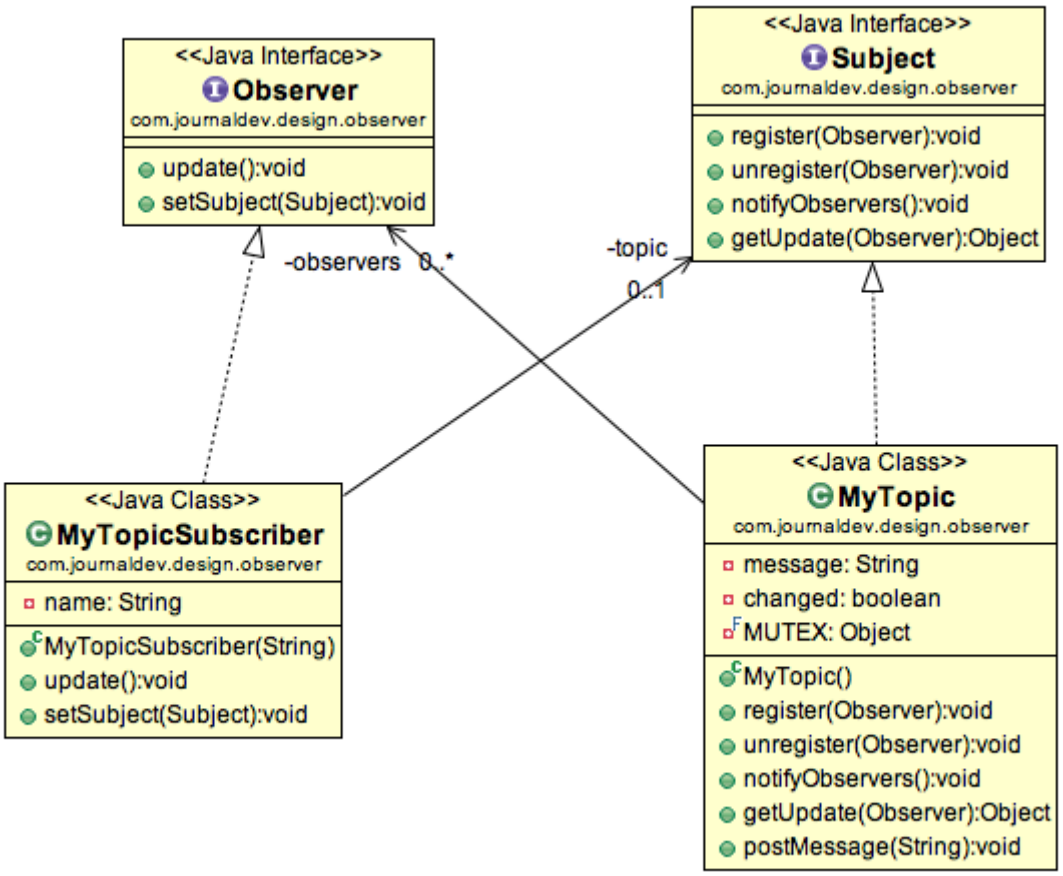
Observer Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó định nghĩa mối phụ thuộc một – nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

Observer có thể đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ thông báo cho Observer biết. Khi không cần nữa, mẫu Observer sẽ được gỡ khỏi hệ thống.

Các thành phần tham gia Observer Pattern:

- Subject : chứa danh sách các observer,  cung cấp phương thức để có thể thêm và loại bỏ observer.
- Observer : định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái.
- Concrete Subject : cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các Concreate Observer, gửi thông báo đến các observer của nó khi có sự thay đổi trạng thái.
- Concrete Observer : cài đặt các phương thức của Observer, lưu trữ trạng thái của subject, thực thi việc cập nhật để giữ cho trạng thái đồng nhất với subject gửi thông báo đến

## 10.2. Class diagram




## 10.3. Build and test

### 10.3.1. Bước 1: Chuẩn bị mẫu

```
package vn.nhannt.observer;

public interface Observer {
   //method to update the observer, used by subject
   public void update();

   //attach with subject to observe
   public void setSubject(Subject sub);
}
```

```
package vn.nhannt.observer;
```

```java
public class MyTopicSubscriber implements Observer{
   private String name;
   private Subject topic;

   public MyTopicSubscriber(String nm){
       this.name=nm;
   }
   @Override
   public void update() {
       String msg = (String) topic.getUpdate(this);
       if(msg == null){
           System.out.println(name+":: No new message");
       }else
           System.out.println(name+":: Consuming message::"+msg);
   }

   @Override
   public void setSubject(Subject sub) {
       this.topic=sub;
   }

}
```

### 10.3.2. Bước 2: Sử dụng mẫu

```java
package vn.nhannt.observer;

public interface Subject {
   //methods to register and unregister observers
   public void register(Observer obj);
   public void unregister(Observer obj);

   //method to notify observers of change
   public void notifyObservers();

   //method to get updates from subject
   public Object getUpdate(Observer obj);
```

```
}
```

```
package vn.nhannt.observer;

import java.util.ArrayList;
import java.util.List;

public class MyTopic implements Subject{
   private List<Observer> observers;
   private String message;
   private boolean changed;
   private final Object MUTEX= new Object();

   public MyTopic(){
       this.observers=new ArrayList<>();
   }
   @Override
   public void register(Observer obj) {
       if(obj == null) throw new NullPointerException("Null Observer");
       synchronized (MUTEX) {
           if(!observers.contains(obj)) observers.add(obj);
       }
   }

   @Override
   public void unregister(Observer obj) {
       synchronized (MUTEX) {
           observers.remove(obj);
       }
   }

   @Override
   public void notifyObservers() {
       List<Observer> observersLocal = null;
        //synchronization is used to make sure any observer registered after
message is received is not notified
       synchronized (MUTEX) {
           if (!changed)
               return;
```

```
            observersLocal = new ArrayList<>(this.observers);
            this.changed=false;
        }
        for (Observer obj : observersLocal) {
            obj.update();
        }

    }

    @Override
    public Object getUpdate(Observer obj) {
        return this.message;
    }

    //method to post message to the topic
    public void postMessage(String msg){
        System.out.println("Message Posted to Topic:"+msg);
        this.message=msg;
        this.changed=true;
        notifyObservers();
    }
}
```

### 10.3.3. Bước 3: Kiểm thử

```
package vn.nhannt.observer;

public class Main {
    public static void main(String[] args) {
        //create subject
        MyTopic topic = new MyTopic();

        //create observers
        Observer obj1 = new MyTopicSubscriber("Obj1");
        Observer obj2 = new MyTopicSubscriber("Obj2");
        Observer obj3 = new MyTopicSubscriber("Obj3");

        //register observers to the subject
```

```
        topic.register(obj1);
        topic.register(obj2);
        topic.register(obj3);

        //attach observer to subject
        obj1.setSubject(topic);
        obj2.setSubject(topic);
        obj3.setSubject(topic);

        //check if any update is available
        obj1.update();

        //now send message to subject
        topic.postMessage("New Message");

    }
}
```

### 10.3.4. Kết quả

```
C:\JavaJDK\v21\bin\java.exe "-javaagent:C:\JetBrains\IntelliJ 2023.3.6\lib\idea
Obj1:: No new message
Message Posted to Topic:New Message
Obj1:: Consuming message::New Message
Obj2:: Consuming message::New Message
Obj3:: Consuming message::New Message

Process finished with exit code 0
```